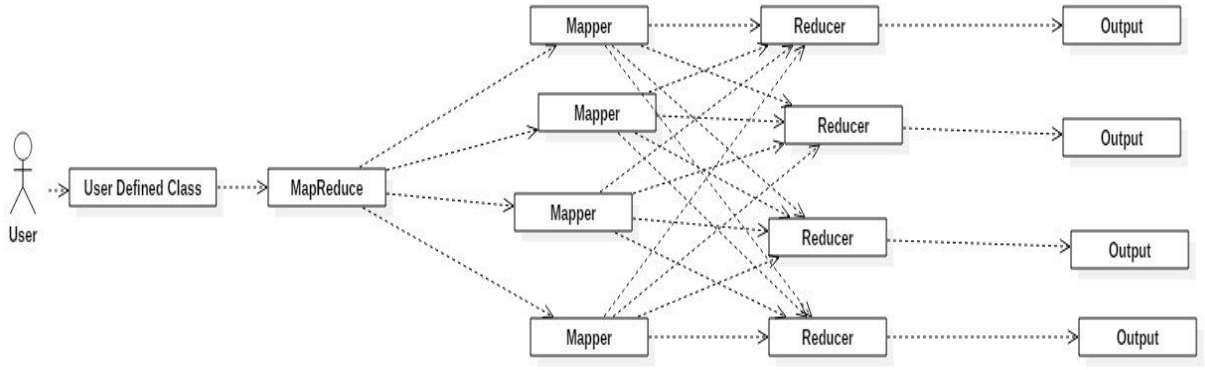
COMPSCI 532 SYSTEMS FOR DATA SCIENCE

- **Course:** COMPSCI 532 Systems for Data Science, Spring 2021
- **Instructor:** Hui Guan
- **Project**: MapReduce
- **Submission**: Mahidhar Kommineni, Sreerag Iyer, Sai Sameer Vennam

Overview

In this project, we have implemented the MapReduce library using Java. The user will be able to write arbitrary user defined Map and Reduce functions which will be compiled along with the source code to be able to perform an executable MapReduce job on a distributed system.

Flow of control:



As seen in the high level architecture above, users will execute the Map Reduce job. The map reduce job will then trigger multiple mapper processes which will invoke the user defined mapper functions on their respective partitions. The reducer processes are triggered after all the mapper processes have completed and the output of each reducer is written to a separate file. Note that the above diagram shows the flow of control, not how the processes communicate. All communication happens between mapper and master and reducer and master.

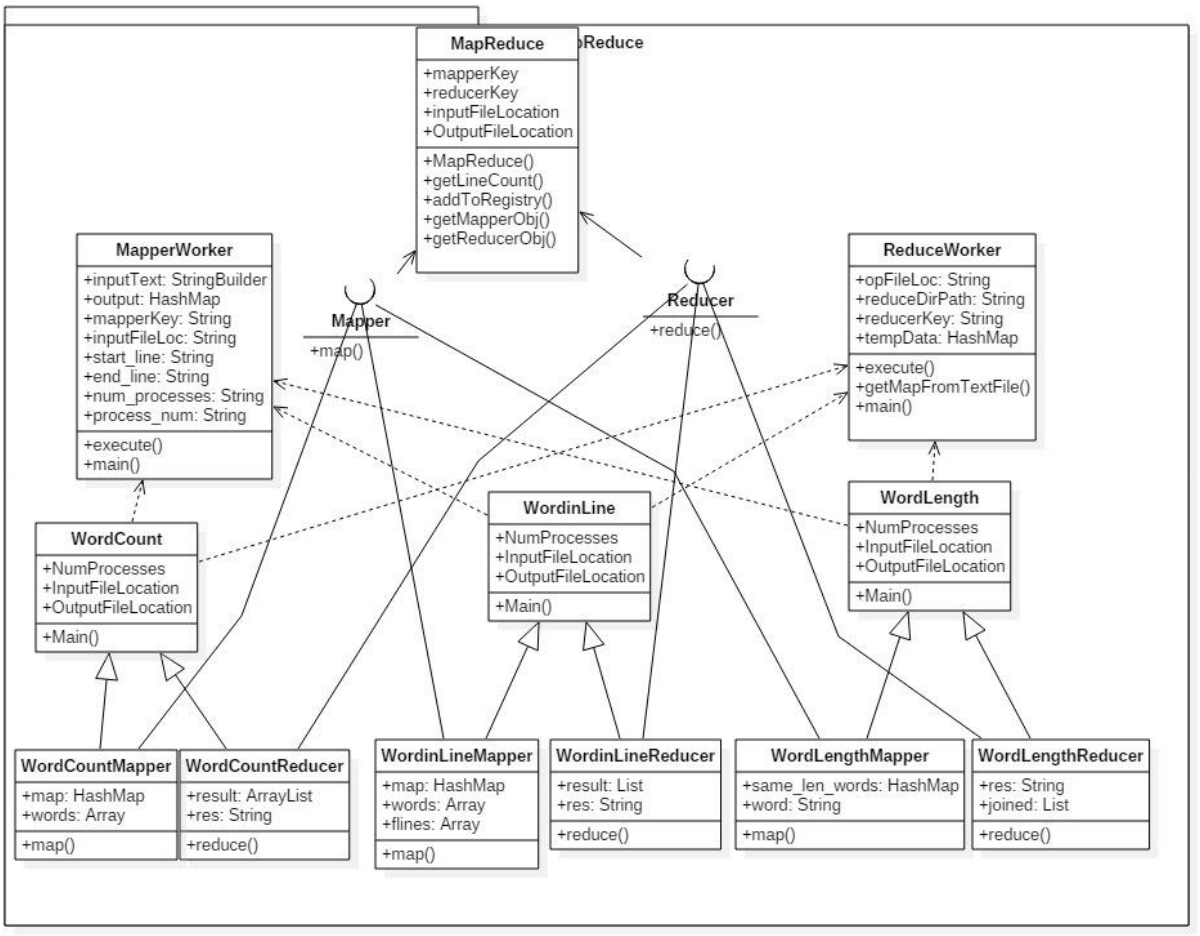
How it works and design considerations:

The master starts each mapper as a new process. Initially, an RMI registry will be started to facilitate communication between the master and the workers. Specifically, the RMI registry will be used by the workers to access the user defined map and reduce objects. The master (src/MapReduce) adds the user defined mapper and reducer objects to the RMI registry. The mapper launches N mapper processes. Each mapper receives an id, number of processes, location of the input file, the starting and ending line numbers of the text file it has to work on. Upon completion of all mapper processes, the master starts N reducer processes that write to N output files.

Distributing the output among intermediate files:

There are N mappers and N reducers. Each mapper produces N intermediate files, corresponding to each of the N reducers (i.e., if there are M mappers and R reducers, each mapper would produce R output files). For example, the first mapper will create a unique intermediate file in each reducer directory - "mapper_0.txt". Reducer N will then read all the files present in the corresponding "reducer_N" directory. This design has the advantage of not requiring the master to explicitly assign intermediate files to reducers. The master only needs to give the locations of the N remote directories "reducer_0", "reducer_1" (and so on) that contain the intermediate files to the reducers. The tradeoff for this is that a total of N*N intermediate files will be formed.

MapReduce UML Class Diagram:



The main Map Reduce class acts as the master worker which will trigger the Mapper worker and Reducer Workers. The mapper and reducer workers will then call the user defined mapper and reducer functions to run the application.

Input parameters to the library

Apart from the user defined map and reduce functions, the user also has to pass certain parameters to the library as input-

- Number of processes (N): This will indicate the N mappers and N reducers
- Input file location: This specifies where the text file on which the MapReduce operation has to be run is present.
- Output file location: This parameter specifies the directory where the N output files from the MapReduce operation will be located.
- Object of the user defined mapper class which will be added to an RMI registry so Master and mapper workers can access it
- Object of the user defined reducer class which will be added to an RMI registry so Master and reducer workers can access it
- Timeout for each worker: If this time limit is exceeded, then the worker stops. This is to mitigate the effect of stragglers.

Function signatures of the map and reduce functions are as follows:

```
HashMap<String,List<String>> map(String,String)throws RemoteException

List<String> reduce(String,List<String>) throws RemoteException
```

Sorting of keys

The mapper computes which intermediate file to write a particular key to by hashing that key to an integer from 0 to N-1. This ensures that all mappers will write to the same intermediate file in their particular remote directory. For example, say mapper 2 and mapper 3 encounter a key “hamlet”, which gets hashed to the number 1 (we use Java’s inbuilt String.hashCode()%N for this). Since this key got hashed to 1, mapper 2 will write it to “mapper_2/reducer_1.txt” and mapper 3 will write it to “mapper_3/reducer1.txt”. So the reduce operation on all occurrences of

the key “hamlet” will be performed by reducer 1. Hence, no shuffling of keys among reducers is required.

Fault tolerance

The following scenarios are considered for fault-

- Worker failure: one particular process got killed and the worker was unable to complete the task.
- Stragglers: One particular worker takes too long to complete.

We simulate fault by randomly selecting one of the N workers (one mapper and one reducer) to be faulty. The master deals with this crash by restarting the failed worker. To mitigate the effect of stragglers, the user has an option to specify a time limit for each worker. By default, this is set to 30 seconds. If the time limit is exceeded, the worker (mapper or reducer) is stopped.

Test Cases

We have taken a total of three user defined test cases upon which we will perform MapReduce jobs. The test cases which contain the business logic would be executed when we run the MapReduce source code. The user defined test cases are WordCount, WordLength and WordinLine. For each of the user defined test cases, there are three classes which are the test class with the main method which runs the entire logic of the test case by calling the MapReduce class in the library, the library will then invoke user defined Mapper and Reducer functions. The user defined mapper functions are invoked to run on each row of the partition assigned to it.The output of this user defined map function is then stored on an intermediate file present on the local file system. This is in the form of a key and a list of values which are associated with each key. After all of the user defined mapper functions are done executing, the work of the reducer begins. Each user defined reducer function handles the information belonging to one key only and then it writes its output to a seperate file. The final output of the MapReduce job would be the grouping of all of the outputs from the reducer.

WordCount

- The test case WordCount gives the number of occurrences of each word present in the file 'hamlet.txt'.
- There are a total of three classes that are present which are WordCount, WordCountMapper and WordCountReducer.

- The WordCount class has the main function which runs MapReduce class which is part of the library to execute the application. We pass the number of processes as ‘4’, the input file location, output file location, the user defined mapper and reducer functions.
- The WordCountMapper class implements the Mapper interface, it has the user defined mapper function and is executed over a portion of the input file. It gets its input in the form of a key and value pair.
- The output of the mapper function is a hashmap which has the word as the key and a list with ‘1’s depending on the number of occurrences of that word. The output is written to intermediate files in the corresponding reducer directory.
- The WordCountReducer Class than aggregates all of the values corresponding to a key and then outputs a single key value pair with word as the key and the count as the value

WordLength

- The test case WordLength returns the word length and all the corresponding words with that length that are present in the file 'lorenipsum.txt'.
- There are a total of three classes that are present which are WordLength, WordLengthMapper and WordLengthReducer.
- The WordLength class has the main function which runs MapReduce class which is part of the library to execute the application. We pass the number of processes as 5, the input file location, output file location, the user defined mapper and reducer functions.
- The WordLengthMapper class implements the Mapper interface, it has the user defined mapper function and is executed over a portion of the input file. It gets its input in the form of a key and value pair.
- The output of the mapper function is a hashmap which has the length of the word as the key and all the list of words with that length. The output is written to intermediate files in the corresponding reducer directory.
- The WordLengthReducer class then reads all intermediate files corresponding to that reducer, aggregates all the words corresponding to a particular length and writes the output in the test case output directory.

WordinLine

- The test case WordinLine returns a word and all the line numbers where that word is present in the input file 'hamlet.txt'.
- There are a total of three classes that are present which are WordinLine, WordinLineMapper and WordinLineReducer.
- The WordLength class has the main function which runs MapReduce class which is part of the library to execute the application. We pass the number of processes as 4, the input file location, output file location, the user defined mapper and reducer functions.

- The WordinLineMapper class implements the Mapper interface, it has the user defined mapper function and is executed over a portion of the input file. It gets its input in the form of a key and value pair.
- The output of the mapper function is a hashmap which has the word as the key and all the list of line numbers where the word has occurred in that partition. The output is written to intermediate files in the corresponding reducer directory.
- The WordLengthReducer class then reads all intermediate files corresponding to that reducer, aggregates all the line numbers corresponding to a particular word and writes the output in the test case output directory.

How to run

The shell script to run all the test cases is present in the root directory ("run-test-cases.sh"). This will compile all the library code with test cases and then run them. The shell script will need to have permissions to run on the user's machine.

```
./run-test-cases.sh
```

To run the spark implementation - the spark implementation is present in the src/spark folder.

```
python3 filename.py
```

Outputs

The MapReduce library will output N files, based on the number of processes. The spark implementation provides one output file.

The output for the test cases are present in the "tes_cases_output" folder. The output for the spark implementation is present in the "spark_output" folder. Both are in the root directory.

Since there is a one fault mapper and one faulty reducer simulated for each case, which leads to a delay of 30 + 30 = 60 seconds for each test case, the total time to run all test cases will be around 3-5 minutes.

System Requirements:

- It has been tested on Mac, Linux (Ubuntu 20.04) and Windows (with git bash).
- Java 8 and above, Python 3+ and Pyspark needs to be installed for running the applications.